FORSCHUNGSZENTRUM JULICH GmbH
Zentralinstitut fur Angewandte Mathematik
D-52425 Julich, Tel. (02461) 61-6402

Interner Bericht

TOP^2 — Tool for Partial Parallelization
Version 3.01
User's Guide

Ulrich Detert, Michael Gerndt

KFA-ZAM-IB-9418

August 1994
(Stand 23.08.94)

TOP^2 - Tool Suite for Partial Parallelization
Version 3.01
User's Guide

Ulrich Detert, Michael Gerndt
Forschungszentrum Jülich GmbH
Zentralinstitut für Angewandte Mathematik
Postfach 1913, D-52425 Jülich

August 23, 1994

Contents

1 Introduction

The parallelization of existing sequential applications for distributed memory parallel systems is often not a trivial task, as parallelization for this type of machines is by its nature not a local operation.

TOP^2 is a tool suite that aids users of such parallel systems in porting existing sequential applications by supporting the separation of compute-intensive kernels of an application from the existing sequential code and providing a development environment for the parallelization of these code segments. Thus, the parallelization of big applications can be broken up into several smaller tasks that may, in a way, be regarded as local optimization steps. In this scenario, the sequential and the parallel code are run simultaneously as a distributed application on both systems and automatically exchange context data between both components. Main features in this process are the provision of cross-domain message passing for the automatic distribution of program data from the sequential machine to the distributed memory system and the ability of on-line debugging of the parallelized code. The input language of TOP^2 is Fortran 77.

The data distribution features of TOP^2 are a subset of those defined in HPF Fortran and thus especially support algorithms on regular data structures exploiting data parallelism in the context of SPMD programming. Logical processor arrays with up to seven dimensions are the basic vehicle for data distribution specifications in this scenario. Fortran data arrays may be distributed by distributing each array dimension onto the corresponding dimension of a processor array. The distribution scheme may be "block", "cyclic", or "replicate" (undistributed) in each dimension. The logical processor arrays are mapped onto the physical processing nodes of the target architecture which implicitely distributes the data onto the nodes. Intrinsic functions for index calculations on distributed arrays are provided in order to facilitate the development of the parallel SPMD code.

TOP^2 consists of two major components, an interactive annotator and a code generator. The annotator supports the user in defining the proper context (input and output variables) of the parallelizable program segment through interprocedural data flow analysis and allows the definition of data distribution strategies. The interactive dialog of the annotator is based on the X Window System. The code generator creates source code for the sequential and the parallel machine that reflects the user-defined data distribution and contains all necessary functions to do the cross-domain message passing for input and output data (Figure 1).

The annotator and the code generator implement two phases of TOP^2. The information exchange between these phases is realized by means of source code directives. The annotator inserts the directives into the given sequential Fortran program according to its data flow analysis and the user interaction. The code generator reads the directives and takes them as a basis for the following code generation phase. As the two phases are only loosely coupled to each other, it is also possible to separate them and e.g. run the code generation phase several times, if changes to the source code directives have been made manually.

At present, two implementations of TOP^2 exist:

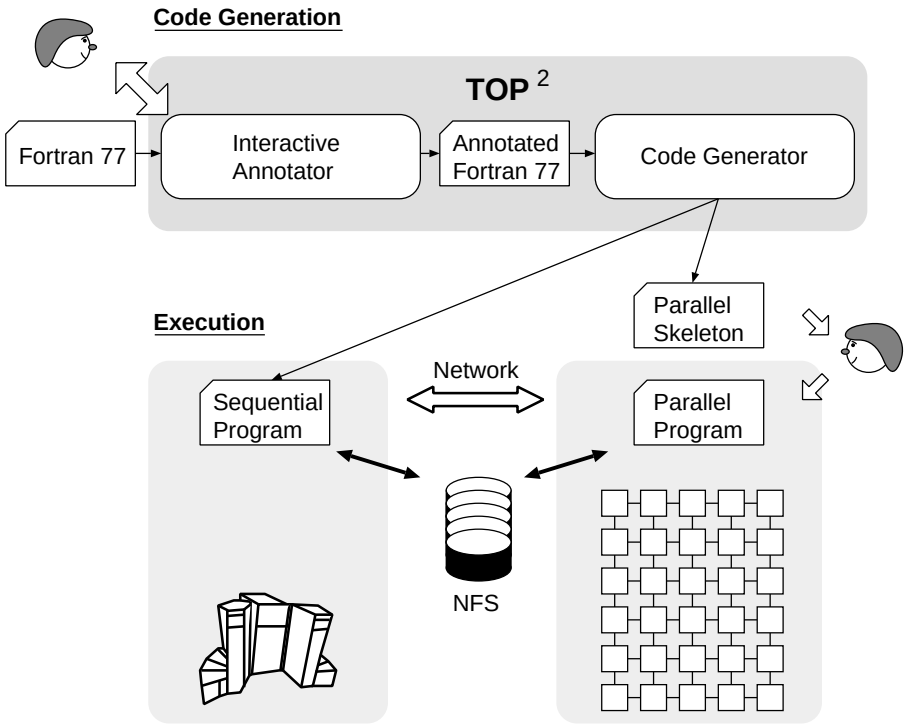

Figure 1: TOP^2 Structure

- At KFA Juelich, TOP^2 is installed on a Sun workstation representing a typical sequential machine and on an Intel Paragon distributed memory system. This is the implementation that is referred to in the main part of this document.
- A prototype implementation of TOP^2 for a Cray T3D system exists at Cray Research Inc. in Eagan/Minnesota. This implementation is referred to in section 13 of this User's Guide.

In the current version of TOP^2, two implementations of the cross-domain message passing exist that may be selected through an option. In the first implementation, data is directly passed via network, using UNIX sockets for communication, in the second, NFS shared files are used for data exchange. Both implementations provide essentially the same functionality, however, differ in performance and portability. Details of the specific installation are described in the following.

2 Quick Start for Impatient Users

To use the current implementation of TOP^2 on the Sun workstation cluster and the Paragon system at KFA (zam127.zam.kfa-juelich.de and paragon.zam.kfa-juelich.de) observe the following guidelines (deviations from this procedure related to the TOP^2 implementation for Cray T3D are described in section 13).

1. Select an existing sequential application for parallelization on the Paragon System. Assure that it runs correctly on the Sun workstation and that all required input data

and pre- and postprocessing procedures exist on the workstation.

2. If possible, establish an NFS shared file system between Sun and Paragon or use an existing one. Create a new directory (called working directory in the following) for your application in this file system and make sure that it has read and write permission on both machines. Make this directory the current directory on both machines for all following steps. (If it is not possible to use an NFS mounted file system on both machines, use two distinct directories on Sun and Paragon. In this case, ftp will have to be used to transport the generated parallel code to the Paragon, and only the network version of the cross-domain message passing may be used for data exchange.)

3. Copy the source of your application into the working directory and, if necessary, also all data or library files that are required for compilation and execution.

4. Make sure that the environment variable `TOP2HOME` has already been set for you by the system administrator or set it yourself to `/usr/local/top2` on Sun and Paragon.

5. From `$TOP2HOME/resources` copy the X Window resource file `Top2` into the working directory (Sun only).

6. Now execute TOP^2 by typing `top2` on the Sun command line to initiate the annotation and code generation phase. During the interaction with TOP^2 execute the following steps:

   (a) Use `Read Program` in the `System` submenu to read the source of your application.

   (b) Click `Load Unit` to select that program unit that forms the top procedure of the program segment that you are about to parallelize.

   (c) From the `Annotation` submenu select `Distributions` to determine and possibly correct or optimize the set of input and output variables. To move a variable from one list to another click on the variable and select one of the buttons `IN`, `INOUT`, or `OUT`.

   (d) In the `Distributions` dialog click `DIST/INOUT` to toggle between the definition of input/output and distribution specifications. To define the distribution specifications of a variable click on the variable and select the desired dimension(s) and data distribution scheme(s).

   (e) If data distribution is to be done onto user-defined logical processor arrays rather than onto the one-dimensional default array, use the function `Processor Arrays` in the `Annotation` submenu to define the name and dimensionality of all processor arrays (use Fortran syntax for the dimensions). Use the `ONTO` clause in the `Distributions` dialog to refer to defined processor arrays.

   (f) The dimensions of processor arrays may be made dependant on symbolic constants. If this is desired, use the names of existing or yet undefined symbolic constants in the processor array dimensions. Specific values for new symbolic constants may be provided via a popup dialog window. The value of existing symbolic constants may be changed with the `Constants` function in the `Annotation` submenu.

   (g) Use the `Options` item in the `Annotation` submenu to switch on debugging, if desired. `First call` and `Last call` may be used to switch on parallelization

(and debugging) for only the specified number of subroutine calls (0 means all calls).

(h) After having made all definitions for input/output, data distribution, and debugging, use one of the functions `MP Transform` or `NFS Transform` in the `System` submenu to activate the code generation phase. Here, `MP` stands for Message Passing via network and `NFS` means data exchange via NFS shared files. For performance reasons and for ease of use, the network communication method should be preferred. Either of these functions will create two source files, one for the modified sequential program and one for the new parallel program. A popup window will inform you about the names of the created source files and eventually about errors that occured during code generation.

(i) Now quit TOP^2 to proceed with the following steps.

7. At this point TOP^2 has generated a sequential program and a parallel program skeleton. The parallel program skeleton contains all code required to do the data distribution and cross-domain message passing. The parallelized program kernel, however, has to be inserted manually. Use the comments in the generated code as a guideline, as to where to insert the parallel code.

8. Compile the generated sequential program part with the command `top2.makes` on the Sun workstation.

9. Compile the parallel program part with the command `top2.makep` either on the Sun workstation (cross-compilation) or on Paragon.

10. The commands `top2.makes` and `top2.makep` implement only basic compiler options for the compilation of the sequential and parallel code. If this is not sufficient, copy the makefile `$TOP2HOME\lib\makefile_sp` into the working directory and modify it according to your needs. The header of the makefile contains information on possible options and targets for compilation.

11. The parallelized program may now be executed as a distributed application on the Sun workstation and on Paragon. Use standard UNIX commands to start the sequential part on the Sun and the `pexec` procedure on the Paragon to start the parallel program part (don't forget to specify the appropriate number of nodes in the `pexec` call). Note that the application must be started from the NFS mounted working directory on both machines, if NFS is used as the data transport mechanism.

12. If debugging is switched on through the appropriate source code directives, you will see messages during program execution on your Sun workstation, if the results of the parallelized program fragment are different between Sun and Paragon (with a certain threshold); otherwise execution will complete silently.

13. If, after initial tests, you want to make changes to the existing application (e.g. change distribution schemes or processor arrays) you will have to rerun TOP^2 and make the desired changes. Note, that previous modifications to the parallel code fragment will not be lost, if all modifications had been made properly in the marked areas. Changes to the generated sequential code or outside the marked areas in the parallel code, however, will be lost and should therefore be avoided.

3 The TOP^2 Annotator

The interactive annotator performs an interprocedural data flow analysis and supports the user in defining the set of input and output variables for the parallel program segment. Secondly, it allows to interactively specify data distribution properties for these variables.

In principle, the interprocedural data flow analysis can be run unattended by the user and will produce correct results, as interprocedural and aliasing effects are taken into account during analysis. In some cases, however, it will be useful to interactively optimize the results of the analysis. This may be the case for array references, if conservative assumptions have been made during the analysis due to the lack of complete def-use and array kill information. An optimization (i.e. reduction in size) of the set of input and output variables will generally lead to better program performance during execution of the generated code, since less data has to be transferred between the sequential and the parallel machine. It is an error, however, to delete variables from the input or output lists that are actually used as input or output, respectively. Variables that are used for the dimensioning of adjustable dimensioned arrays e.g. need to be declared in the input list.

The second task of the annotator, the definition of data distribution, is always a user-driven interactive process. The TOP^2 annotator provides dialogs to easily select arrays in the input/output set and define distribution strategies for them. Furthermore, logical processor arrays can be defined as the target of such data distributions.

Another interactively supported action is the definition of symbolic constants that may be used in the specification of processor arrays and can help to make the program code mostly independent of a specific choice of the number of physical processors.

3.1 Interactive functions

In the following, all functions of the annotator will be shortly described:

Submenu "System":

Load Unit: Load a program unit into TOP^2 for further processing. This call must be preceeded by a call to `Read Program` in either the same session or a previous session.

Read Program: Read the source of a Fortran program from a file and convert it into TOP^2 internal representation. The source program must be given as a single file. For later processing the program is split up into individual program units which are stored in the subdirectory `units` of the working directory.

NFS Transform: Initiate the code generation phase of TOP^2 with data communication being implemented via NFS shared files. This function produces a Fortran program annotated with directives and calls the TOP^2 code generator which will generate a modified sequential program and a parallel program skeleton out of the annotated program. The naming conventions for the generated source files are as follows: *name*`.top2.f` is the name of the annotated program, *name*`.seq.f` is the name of

the sequential and *name*`.par.f` the name of the parallel program generated by the code generator. In all cases *name* stands for the name of the original input program. Details of the code generation phase are described in section 4.

MP Transform: This function is equivalent to the function `NFS Transform`, with the only exception that `MP Transform` generates code that implements data communication via network, rather than via NFS shared files.

Quit: Exit TOP^2.

Submenu "Annotation":

Processor Arrays: Define logical processor arrays for later use in data distribution specifications through a popup dialog window. The dialog allows to either enter the specification of new processor arrays or modify those of existing ones. Processor arrays may be one- to seven-dimensional. The processor array specifications may be entered by the user in Fortran-like syntax where the dimensions may be either constants or symbolic constants (e.g. `PROC(12,ip,ir)`). If processor array dimensions are defined through symbolic constants, a separate popup dialog allows to enter specific values for these constants. During execution of the parallelized application, each processor array is mapped to the physically available processors. Therefore, the size of all processor arrays (product of all dimensions) must be identical and must match the number of processors allocated during program execution.

Constants: Modify the value of already existing symbolic constants. Symbolic constants are implemented through Fortran `PARAMETER` statements.

Distributions: Modify the input/output lists generated by the annotator and define data distribution strategies for arrays in these lists. The button `DIST/INOUT` in the popup dialog window allows to toggle between these two functions. If the input/output function is activated, variables and arrays may be moved from one list to another by clicking on them and selecting the desired list. If the distribution function is activated, distributions may be defined by clicking on the variable and selecting the desired distribution policy in the appearing popup window. Block, cyclic, or replicated ("`*`") distribution may be specified for each of the array dimensions. In this dialog the `ONTO` clause may be used to define distribution onto an already defined processor array. If this feature is used, the number of the processor array dimensions must match the number of those array dimensions with distribution policy other than "replicate".

If multiple arrays are selected in the dialog window one after the other, the following selection of the distribution strategy and the `ONTO` clause will be applied equally to all selected arrays. This mode of operation may be useful, if a number of arrays require identical data distributions.

Options: Switch on debugging. `First call` and `Last call` may be used to switch on parallelization (and debugging) for only the specified number of subroutine calls (0 means all calls). The latter functions may be used to reduce the debugging overhead and the overhead for distributed processing of the application, if it is known e.g. that an error that is to be analyzed occurs only after a certain number of calls to the parallelized subroutine.

3.2 Resources

The TOP^2 annotator uses an X Window resource file named `Top2` that defines several properties of dialog items. It is recommended that this resource file be copied from `$TOP2HOME/resources` into the working directory to assure that it is read during start-up of TOP^2, or that the standard X Window function `xrdb` be used to merge the contents of the file into the existing X server's resource database. If desired, user-specific changes may be applied to this file.

4 TOP^2 Code Generator

The TOP^2 code generator converts the source directives inserted into the input program into a sequential program and a parallel program skeleton. The sequential program is complete in that it may be compiled without further changes. The parallel program skeleton, however, needs to be completed by the programmer, as it doesn't yet contain the code for the parallel program segment. Details on how the generated code is structured and what modifications are to be made by the user are given in section 8.

The code generator is normally called by the annotator, but may as well be called directly from the command line. It takes some options that may either be specified on the command line or be provided through a resource file and the name of the annotated input file. Modifications applied to the resource file will be honored independent on how the code generator was called, command line options may, however, only be set, if it is called from the command line.

4.1 Command Line Call

If the code generator is called from the command line, the general syntax of the call is as follows:

`top2pp` [*options*] *filename*

> where *filename* is the name of the annotated Fortran program and *options* may be of the following:

`-e`[`ewci`] or `-d`[`ewci`]

> `-e` switches on, `-d` switches off messages of class `e` (error), `w` (warning), `c` (caution), `i` (informative). The default is `-eewci`.

`-l`[`a`|`c`|`i`]

-l defines the language dialect to be used: a (ANSI), c (CRAY), i (IBM). This switch is used for the proper handling of data types in declarations and cross-domain message passing. For byte-oriented machines like workstations the switch should be set to `-li` (default for the Sun implementation).

`-c[c|x]`

-c defines the communication method to be used for data exchange between the sequential and the parallel program segment: `c` denotes communication via NFS shared files and `x` cross-domain message passing via network (UNIX socket communication). In both cases, XDR routines are used for data conversion. The default is `-cx`.

4.2 Resources

The resource file of the code generator allows to define user-specific options, even if the code generator is called by the annotator. A default resource file is provided in `$TOP2HOME/resources`, named `top2pprc`. This file may be copied into a user-owned directory and modified according to the user's needs. In order to activate a resource file other than the default file, the environment variable `TOP2PPRC` has to be set to the new filename (absolute or relative path name including the file name).

As the code generator is not an X Window based application, the resource file is not X Window specific and may not be merged into the X server resource data base. The general syntax of entries in the resource file is as follows:

resourcename resourcevalue

The following name/value pairs are currently recognized:

`N$PROC` *number*

Define a default number of processing nodes for the parallel program for use in default processor arrays (default: 4).

`SEQ_IN_FILE` *filename*

Name prefix for the files containing the input data for the parallel program segment during cross-domain communication. This name prefix is used by the sequential program segment to write the "input" data. For each processing node the file prefix is suffixed with the node number. "`IN.`" is the default.

`PAR_IN_FILE` *filename*

Name prefix for the files containing the input data for the parallel program segment during cross-domain communication. This name prefix is used by the parallel program segment to read the "input" data. For each processing node the file prefix is suffixed with the node number. For communication via NFS shared files, this file name should be identical to the name defined under `SEQ_IN_FILE`. "`IN.`" is the default.

`SEQ_OUT_FILE` *filename*

Name prefix for the files containing the output data of the parallel program segment during cross-domain communication. This name prefix is used by the sequential program segment to read the "output" data. For each processing node the file prefix is suffixed with the node number. "`OUT.`" is the default.

`PAR_OUT_FILE` *filename*

Name prefix for the files containing the output data of the parallel program segment during cross-domain communication. This name prefix is used by the parallel program segment to write the "output" data. For each processing node the file prefix is suffixed with the node number. For communication via NFS shared files, this file name should be identical to the name defined under `SEQ_OUT_FILE`. "`OUT.`" is the default.

`MSG_FILE` *filename*

Name of the file receiving all messages of the code generator. Default is `stdout`.

`MSG_ERROR` *boolean*
`MSG_WARNING` *boolean*
`MSG_CAUTION` *boolean*
`MSG_INFO` *boolean*

Switch on or off messages of severity levels `Error` to `Informative`. Default is `true` for all messages.

`PRETTY_PRINTER` *boolean*

Switch on or off the usage of the Fortran pretty printer for all automatically generated code. Default is `true`.

`P_MACHINE` *netname*

Define the name of the parallel machine in the network. The default is `Paragon`.

`COMMUNICATION` *method*

Define the communication method for the exchange of data between the sequential and the parallel machine. *Method* may be either `NETWORK` (default) or `NFS`.

`CMD_PORT` *number*

Define the port number for initial communication setup, if the selected communication method is `NETWORK`. Note that this port is only used for communication setup, the actual data exchange is realized via dynamically allocated ports. *Number* must be an integer in the range $5000 \leq number \leq 16383$. The default is `5011`.

5 Source Code Directives

Source code directives are used for communication between the TOP^2 annotator and the code generator. Normally there will be no need for the user to directly insert or modify directives, even though this is well possible. In the following, syntax and semantics of all directives will be described.

The general format of TOP^2 directives is

`CKFA$` *keyword* [*value*]

Each directive starts on a new line at column one. If necessary, directives may be continued on subsequent lines in the format

`CKFA$*` *continued_text*,

however, all keywords must appear on the first line. TOP^2 source code directives are not case sensitive.

The following keyword/value pairs are currently recognized by the code generator:

`IN` *variable_list*
`OUT` *variable_list*
`INOUT` *variable_list*

These directives define variables and arrays that are input or output for the parallel program segment. *Variable_list* is a blank- or comma-separated list of variable names.

Example:

```
CKFA$ IN A, SUM, CARRAY
```

PROCESSORS *procspec_list*

This directive defines one- or multi-dimensional logical processor arrays that can be used for data distribution. *Procspec_list* is a blank- or comma-separated list of processor specifications. Each processor specification has the form *procname*(*dim*[,*dim* ...]), where *procname* is the name of the processor array and *dim* is a dimension expression consisting of either a constant or a symbolic constant (Fortran PARAMETER constant). A maximum of seven dimensions may be specified for each processor array. The size of each processor array (the product of all dimensions) directly determines the number of compute nodes used during execution. If more than one processor array is defined, the sizes of all processor arrays must be equal.

Example:

```
CKFA$ PROCESSORS P(2,IP), Q(16), RS(2,2,4)
```

DISTRIBUTE *distspec_list*

The distribute directive defines the data distribution policy for one or more arrays. *Distspec_list* is a list of distribution specifications, each defining the distribution policy for one array. The format of each *distspec* item is defined as follows:

distspec := *arrayspec* [*ontoclause*]

arrayspec := *arrayname*(*dspec*[,*dspec* ...])

dspec := BLOCK | B | CYCLIC | C | *

ontoclause := ONTO *procname*

"B" and "C" are abbreviations for "BLOCK" and "CYCLIC", respectively, "*" means replication in that dimension. Note that the ONTO clause may only be missing, if distribution is to be done in only one dimension onto the one-dimensional default processor array. More generally, the number of distributed array dimensions (those different from "*") must match exactly the number of dimensions of the corresponding processor array.

Examples:

```
CKFA$ DISTRIBUTE A(BLOCK,CYCLIC,*) ONTO P
CKFA$ DISTRIBUTE B(*,B,*) ONTO Q, D(C) ONTO Q
CKFA$ DISTRIBUTE CARRAY(BLOCK)
```

N$PROC *number*

In this directive *number* defines the number of processing nodes used for data distribution; this is also the size of the one-dimensional default processor array

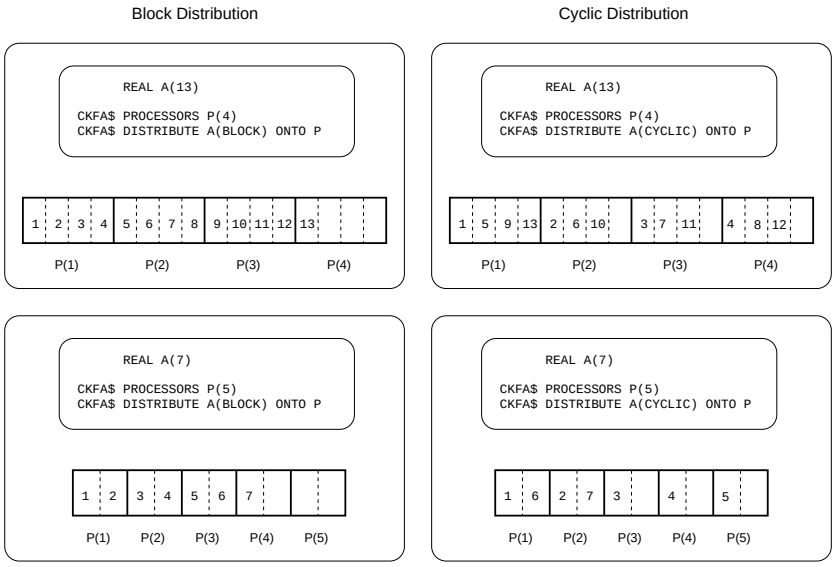

Figure 2: Example - Block and Cyclic Distribution

that is used if no `ONTO` clause is specified. Note that the specified *number* must be consistent with the size of all defined processor arrays. *Number* may be either a constant or a symbolic constant. If the `N$PROC` directive is not specified, the number of processing nodes is computed from the size of any of the logical processor arrays. If no `PROCESSORS` directive is defined, a TOP^2 specific default is chosen (compare section 4.2).

Example:

```
CKFA$ N$PROC 32
```

`DEBUG`

This directive switches on debugging.

`STARTSTOP` *start* [*stop*]

This directive allows to restrict parallelization of the selected parallel program segment to a specific number of calls of the parallelized subroutine. *Start* is an integer number specifying the first call that is to be executed in parallel, *stop* specifies the call after which processing is to be stopped. Before *start* is reached, processing of the application is merely sequential, if *stop* is reached, processing halts. A value of 0 for *start* means that parallel processing is started right from the beginning (default), a value of 0 for *stop* means that processing is not to be halted before the completion of the program (default).

6 Details of Data Distribution

For the development of the parallel code it will be necessary to precisely know how data items are distributed onto the physical processing nodes. Two important basic terms in this respect are "block" and "cyclic" distribution. Fig. 2 gives examples of these distribution schemes for one-dimensional arrays. From the performance point of view, it is important to notice that block distribution may lead to significant load imbalance, if the array dimensions are not divisible by the number of blocks. Cyclic distribution, on the other hand, can lead to an imbalance of one element per block at maximum.

The extension of one-dimensional distributions to multi-dimensional distributions is obtained by distributing each array dimension onto one dimension of a multi-dimensional processor array, where the number of distributed array dimensions must match the number of processor array dimensions. The processor array is then mapped onto the physical compute nodes in column major order. Fig. 3 gives an example of a two-dimensional distribution for a three-dimensional array.

For most applications it will not be necessary to directly refer to the formulae given below, since all basic index calculations for distributed arrays can be realized by means of the appropriate intrinsic functions that are available at run-time in the parallel code (see section 7 for details).

In the following, details of processor mappings and index calculations for multi-dimensional distributions will be given.

6.1 Mapping Logical Processor Arrays to Physical Processors

Let $P := \{(r_1, r_2, \ldots, r_s) \mid 1 \leq r_i \leq p_i\}$ be an s-dimensional processor array with

$$n := \prod_{i=1}^{s} p_i \tag{1}$$

being the number of physical processing nodes. p_i is called the size of dimension i of the processor array.

Let $(r_1, r_2, \ldots, r_s) \in P$ be a logical processor with respect to processor array P. The mapping $\mathcal{P}_P : P \to [0, \ldots, n-1]$ with

$$\mathcal{P}_P(r_1, \ldots, r_s) = \sum_{i=1}^{s} (r_i - 1) \prod_{k=1}^{i-1} p_k \tag{2}$$

defines the column major ordering of logical processors in P onto the physical processors.

The inverse $\overline{\mathcal{P}}_P : [0, \ldots, n-1] \to P$, mapping physical processors to logical processors in processor array P, is given by $\overline{\mathcal{P}}_P(n) = (r_1, \ldots, r_s)$ with

$$r_i = \left\lfloor \frac{n \bmod \prod_{k=1}^{i} p_k}{\prod_{k=1}^{i-1} p_k} \right\rfloor + 1, \quad 1 \leq i \leq s. \tag{3}$$

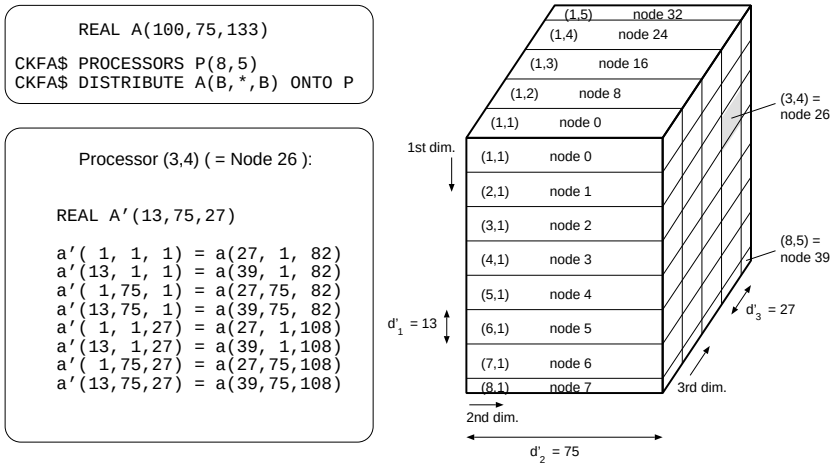

Figure 3: Example - Block Distribution of 3-dimensional Array

6.2 Mapping Array Elements to Logical Processors

Let $A := \{(a_1, a_2, \ldots, a_m) \mid l_i \leq a_i \leq u_i\}$ be (the index space of) an m-dimensional Fortran data array with $d_i := u_i - l_i + 1$ (l_i lower bound, u_i upper bound, d_i size of dimension i).

Let P be an s-dimensional processor array with $s \leq m$.

The m-tuple $D_P^A := (g_1, g_2, \ldots, g_m)$ is called a distribution of A with respect to P if the following holds:

$$g_i \in \{\text{block}, \text{cyclic}, \text{replicate}\}$$

and

$$g_i \neq \text{replicate}, \quad \text{for exactly } s \text{ elements.}$$

For the following, those elements $g_i \neq$ replicate will be numbered $g_{j_1}, g_{j_2}, \ldots, g_{j_s}$, such that g_{j_k} corresponds to dimension k of processor array P.

D_P^A defines a mapping $\mathcal{D}_P^A : A \to P \times A'$ of elements of the undistributed array A to elements of the distributed array A' on logical processor $(r_1, \ldots, r_s) \in P$ with

$$\mathcal{D}_P^A(a_1, \ldots, a_m) = (r_1, \ldots, r_s, a'_1, \ldots, a'_m).$$

It is $A' := \{(a'_1, \ldots, a'_m) \mid l'_i \leq a'_i \leq u'_i\}$ and $d'_i := u'_i - l'_i + 1$ with

$$d'_i = \begin{cases} \left\lceil \frac{d_i}{p_k} \right\rceil & \text{if } g_i \neq \text{ replicate} \\ d_i & \text{else} \end{cases} \tag{4}$$

where p_k corresponds to element $g_i = g_{j_k}$.

In TOP^2 the lower bound of any distributed array A' is always identical to the lower bound of the corresponding undistributed array A, i.e.

$$l'_i = l_i \tag{5}$$

and thus

$$u'_i = d'_i + l_i - 1. \tag{6}$$

The index mapping of elements of A to elements of A' (i.e. the mapping of global array indices to local indices) is given by

$$a'_i = \begin{cases} ((a_i - l_i) \bmod d'_i) + l'_i & \text{if } g_i = \text{block} \\ \lfloor (a_i - l_i)/p_k \rfloor + l'_i & \text{if } g_i = \text{cyclic} \\ a_i & \text{if } g_i = \text{replicate} \end{cases} \tag{7}$$

The logical processor number $(r_1, \ldots, r_s)$ is given by

$$r_k = \begin{cases} \lfloor (a_{j_k} - l_{j_k})/d'_{j_k} \rfloor + 1 & \text{if } g_{j_k} = \text{block} \\ ((a_{j_k} - l_{j_k}) \bmod p_k) + 1 & \text{if } g_{j_k} = \text{cyclic} \end{cases} \tag{8}$$

The inverse $\overline{\mathcal{D}}_P^A : P \times A' \to A$, mapping local indices to global indices, is given by $\overline{\mathcal{D}}_P^A(r_1, \ldots, r_s, a'_1, \ldots, a'_m) = (a_1, \ldots, a_m)$ with

$$a_i = \begin{cases} d'_i(r_k - 1) + (a'_i - l'_i) + l_i & \text{if } g_i = \text{block} \\ (a'_i - l'_i)p_k + (r_k - 1) + l_i & \text{if } g_i = \text{cyclic} \\ a'_i & \text{if } g_i = \text{replicate} \end{cases} \tag{9}$$

where k corresponds to $i = j_k$.

7 Intrinsic Functions for Index Calculations on Distributed Arrays

TOP^2 supports the partial parallelization of big applications by providing a mechanism for the separation of parallelizable code segments from the sequential code. This includes an easy way to experiment with various data distribution schemes via interactive user directives. As, however, the resulting parallel SPMD code has to be provided manually by the user, it may be rather intricate to implement the index calculations for the addressing of distributed arrays, if complicated distribution schemes have been used in the interactive phase of TOP^2.

In the following, a set of intrinsic functions will be described that facilitate index calculations on distributed arrays. This set of routines also includes functions for the inquiry of array, processor array, or distribution properties in the parallel code. The described functions are 'intrinsic' in that they are directly related to the distribution scheme chosen during the interactive phase of TOP^2, i.e. information from this phase is automatically passed to the intrinsic functions.

In order to allow the utilization of the TOP^2 intrinsic functions even if an application has been entirely parallelized and ported to the parallel machine (and, thus, should be independent of TOP^2), the implementation of these functions follows a two-level procedure:

- Level-one routines collect all information on arrays, processor arrays, and distributions and store them in an internal data base. As long as TOP^2 is used, these routines are automatically called in the parallel program for all relevant distribution items (compare section 7.3). If the TOP^2 intrinsics are to be used separate from TOP^2, however, the level-one routines have to be explicitely called in the user code.

- Level-one routines return a handle for each array, processor array, and distribution that is defined. These handles are passed to the level-two routines that implement the actual index calculations for the thus identified items. Handles may be passed through a predefined COMMON block `/TOP2_XQ5/` within the user code, if TOP^2 is used, or through a user-defined COMMON block, if used independently of TOP^2.

  When used with TOP^2, the naming convention for handles is such that TOP^2 appends a suffix `_phd$` to each processor name (thus a processor array PROC may be identified through the handle `PROC_phd$`). Respectively, `_ahd$` is appended to each array name and `_dhd$` to the name of each array distributed onto a given processor array.

7.1 Level-One Intrinsic Functions

Level-one routines define or un-define the size and shape of arrays and processor arrays or the characteristics of distribution schemes used for the distribution of arrays onto processor arrays. All arguments described in the following are of type INTEGER.

Define the size and shape of a processor array:

```
CALL TOP2_DEF_PROC(proc_phd$,dim,d)
```

`proc_phd$`	processor array handle (output)
`dim`	number of dimensions (input)
`d`	1-D array of size 7 holding the processor array sizes (input)

Define the size and shape of a Fortran array:

```
CALL TOP2_DEF_ARRAY(array_ahd$,dim,bounds)
```

`array_ahd$`	array handle (output)
`dim`	number of dimensions (input)
`bounds`	1-D array holding the lower and upper bounds of the array in the form low1, high1, low2, high2, ... (input)

Define the distribution of an array onto a processor array:

```
CALL TOP2_DEF_DIST(dist_dhd$,proc_phd$,array_ahd$,dim,d)
```

`dist_dhd$`	distribution handle (output)
`proc_phd$`	processor array handle (input)
`array_ahd$`	array handle (input)

`dim`	number of dimensions (input)
`d`	1-D array of size 7 holding the distribution characteristics (input): 0 = replicate (undistributed) 1 = block 2 = cyclic

Undefine one or all defined processor array(s), i.e. release the processor array handle(s):

```
CALL TOP2_UNDEF_PROC(proc_phd$)
```

`proc_phd$`	processor array handle, if specified as -1 then all currently assigned processor array handles are released (input)

Undefine one or all defined Fortran array(s), i.e. release the array handle(s):

```
CALL TOP2_UNDEF_ARRAY(array_ahd$)
```

`array_ahd$`	array handle, if specified as -1 then all currently assigned array handles are released (input)

Undefine one or all defined array distribution(s), i.e. release the distribution handle(s):

```
CALL TOP2_UNDEF_DIST(dist_dhd$)
```

`dist_dhd$`	distribution handle, if specified as -1 then all currently assigned distribution handles are released (input)

7.2 Level-Two Intrinsic Functions

Level-two routines implement index calculations and inquiry functions for items defined through level-one routines and identified through the appropriate handle. In the following, the term global with respect to indices denotes indices in the undistributed array as existing in the sequential program, the term local, on the other hand, denotes compute node specific indices of a distributed array. All arguments described in the following are of type INTEGER.

Compute the physical node number of the owner of an array element identified by its global array indices:

```
CALL TOP2_GET_OWNER(dist_dhd$,indx,owner)
```

`dist_dhd$`	distribution handle of distributed array (input)
`indx`	1-D array holding the global array indices (input)
`owner`	physical node number of owner (output)

Compute the local indices of a distributed array from its global indices:

```
CALL TOP2_GET_LOCAL(dist_dhd$,gx,lx)
```

`dist_dhd$`	distribution handle of distributed array (input)
`gx`	1-D array holding the global array indices (input)
`lx`	1-D array taking the local array indices (output)

Compute the global indices from the local indices of an array element for a given owner:

```
CALL TOP2_GET_GLOBAL(dist_dhd$,owner,lx,gx)
```

`dist_dhd$`	distribution handle of distributed array (input)
`owner`	physical node number of owner (input)
`lx`	1-D array taking the local array indices (input)
`gx`	1-D array holding the global array indices (output)

Compute the range of global indices of an array section owned by a given owner:

```
CALL TOP2_GET_RANGE(dist_dhd$,owner,low,high,inc)
```

`dist_dhd$`	distribution handle of distributed array (input)
`owner`	physical node number of owner (input)
`low`	1-D array taking the lower bounds of the global array indices in each dimension (output)
`high`	1-D array taking the upper bounds of the global array indices in each dimension (output)
`inc`	1-D array taking the increment of the global array indices in each dimension (for cyclic distribution) (output)

Compute the size and shape of a local array section for a given owner (this results in the actually used array elements, not the declared dimensions of the local array):

```
CALL TOP2_GET_SHAPE(dist_dhd$,owner,low,high)
```

`dist_dhd$`	distribution handle of distributed array (input)
`owner`	physical node number of owner (input)
`low`	1-D array taking the lower bounds of the local array indices in each dimension (output)
`high`	1-D array taking the upper bounds of the local array indices in each dimension (output)

Compute the logical processor array element indices in a processor array (corresponding to a given processor handle) from the physical node number:

```
CALL TOP2_GET_PROC(proc_phd$,node,indx)
```

`proc_phd$`	processor array handle (input)
`node`	physical node number (input)
`indx`	1-D array taking the processor array indices (output)

Compute the physical node number from the processor array indices of a given logical processor array:

```
CALL TOP2_GET_NODE(proc_phd$,indx,node)
```

`proc_phd$`	processor array handle (input)
`indx`	1-D array holding the processor array indices (input)
`node`	physical node number (output)

Compute all physical node numbers for a given logical processor array and store them in an array conformant in size and shape to the processor array:

```
CALL TOP2_GET_ALLNODES(proc_phd$,nodes)
```

`proc_phd$`	processor array handle (input)
`nodes`	n-D array taking the node numbers for the given processor array (output)

Compute the size and shape of a logical processor array as defined in a previous call to `TOP2_DEF_PROC`:

```
CALL TOP2_GET_PSHAPE(proc_phd$,dim,indx)
```

`proc_phd$`	processor array handle (input)
`dim`	processor array dimension (output)
`indx`	1-D array taking the processor array size for each dimension (output)

Compute the global size and shape of an undistributed array as defined in a previous call to `TOP2_DEF_ARRAY`:

```
CALL TOP2_GET_ARRSHAPE(arr_ahd$,dim,low,high)
```

`arr_ahd$`	array handle (input)
`dim`	array dimension (output)
`low`	1-D array taking the array's lower bounds for each dimension (output)
`high`	1-D array taking the array's upper bounds for each dimension (output)

Compute the characteristics of a distribution as defined in a previous call to `TOP2_DEF_DIST`:

```
CALL TOP2_GET_DIST(dist_dhd$,proc_phd$,arr_ahd$,dim,dist)
```

`dist_dhd$`	distribution handle (input)
`proc_phd$`	processor handle (output)
`arr_ahd$`	array handle (output)
`dim`	number of dimensions of the distribution (output)
`dist`	1-D array taking the distribution specifier for each dimension: 0 = replicate, 1 = block, 2 = cyclic. (output)

7.3 Example

The following section gives a short example for the use of the TOP^2 intrinsic functions in a parallel program. As denoted below, part of the code stems from the interactive annotation phase of TOP^2, some parts are automatically generated during the code generation phase of TOP^2, and some have to be provided by the user (compare section 8 for details on the structure of the generated code).

The following example assumes three directives to be created during the interactive phase of TOP^2:

```
CKFA$ PROCESSORS P(IP1,IP2)
CKFA$ INOUT ARRAY
CKFA$ DISTRIBUTE ARRAY(BLOCK,CYCLIC,*) ONTO P
```

The parallel main program produced during the code generation phase of TOP^2 will contain all code required for the definition of the distributed arrays and the involved processor arrays. All handles of these items will be declared and defined, and will be passed in a COMMON block named `/TOP2_XQ5/` for use in the user-provided parallel subroutine (Note that the one-dimensional default processor array `P$PROC` and its corresponding processor array handle `P$PROC_phd$` are always defined):

```
      PROGRAM PARMAIN

      . . .

      COMMON /TOP2_XQ5/ P$PROC_phd$, P_phd$, ARRAY_dhd$, ARRAY_ahd$
      SAVE /TOP2_XQ5/
C
C --- Define processor descriptors ---
C
      P$PROC_$pd$(1) = 1
      P$PROC_$pd$(2) = 8
C
      P_$pd$(1) = 2
      P_$pd$(2) = 4
      P_$pd$(3) = 2

      . . .

      CALL TOP2_DEF_PROC(P$PROC_phd$, P$PROC_$pd$(1), P$PROC_$pd$(2))
      CALL TOP2_DEF_PROC(P_phd$, P_$pd$(1), P_$pd$(2))
      CALL TOP2_DEF_ARRAY(ARRAY_ahd$, ARRAY_$ad$(4), ARRAY_$ad$(5))
      CALL TOP2_DEF_DIST(ARRAY_dhd$, P_phd$, ARRAY_ahd$, ARRAY_$dd$(1),
     1                   ARRAY_$dd$(2))

      CALL PARSUB(ARRAY)

      . . .
```

The generated parallel subroutine skeleton contains code that makes the handles of all distribution items available in `COMMON /TOP2_XQ5/`. Declarations that might be required for arguments to any of the TOP^2 intrinsic functions, however, must be specified in the user-specific code section:

```
      SUBROUTINE PARSUB(ARRAY)

      PARAMETER (N1=100, N2=100, N3=10)
```

```
      REAL*8 ARRAY((N1-1+4)/4,(N2-1+2)/2,N3)

      . . .
C
C --- Common block for address intrinsics ---
C
      COMMON /TOP2_XQ5/ P$PROC_phd$, P_phd$, ARRAY_dhd$, ARRAY_ahd$
      SAVE /TOP2_XQ5/
      INTEGER P$PROC_phd$, P_phd$, ARRAY_dhd$, ARRAY_ahd$
C
C --- PARSUB: --- Make all your code changes below this line ---
C
      integer low(7),high(7),owner,indx(7),lx(7)
C
C  Use TOP2 intrinsics to print global array element
C  ARRAY(73,25,3) on the local node
C
      indx(1) = 73
      indx(2) = 25
      indx(3) = 3

      call top2_get_owner(ARRAY_dhd$,indx,owner)

      if (owner .eq. mynode()) then
        call top2_get_local(ARRAY_dhd$,indx,lx)
        write(*,*) 'ARRAY(73,25,3) = ', ARRAY(lx(1),lx(2),lx(3))
        write(*,*) 'Owner = ',owner,' local = ', lx(1),lx(2),lx(3)
      endif
C
C  Use TOP2_GET_SHAPE to compute the bounds of each array
C  section on the node
C
      call top2_get_shape(ARRAY_dhd$,mynode(),low,high)

      do i = low(1), high(1)
        do j = low(2), high(2)
          do k = low(3), high(3)
            array(i,j,k) = array(i,j,k) + 1.0
          enddo
        enddo
      enddo

      . . .
```

8 The TOP^2 Generated Code

The source code generated by TOP^2 is basically structured into two independent components, the sequential code, containing practically the entire code from the original

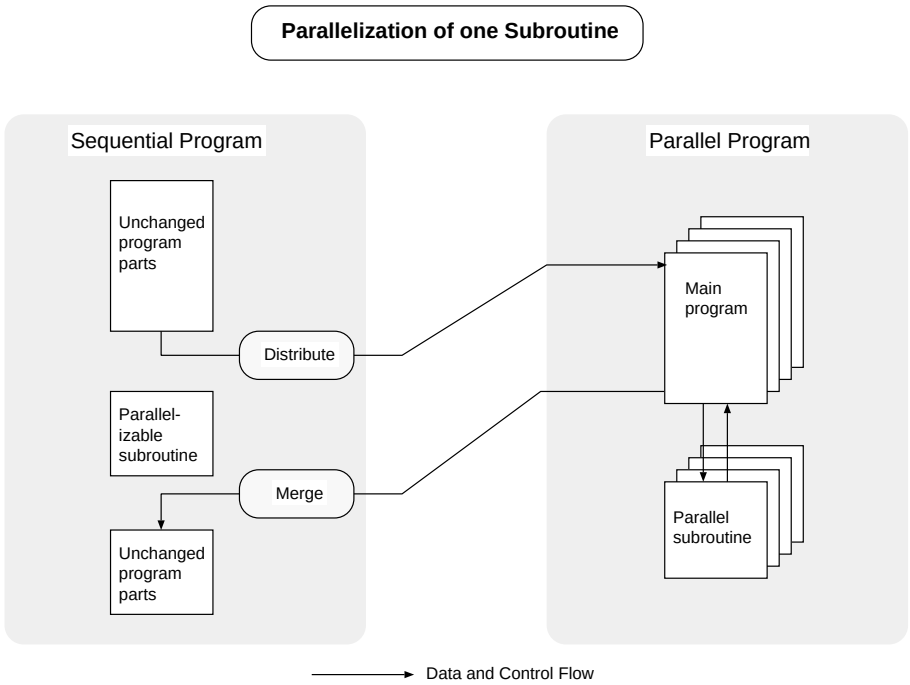

Figure 4: Structure of TOP^2 Generated Code

application, and the parallel code skeleton, containing all declarations required for the parallelization of the selected program kernel and code for the cross-domain message passing (Figure 4).

Normally, there will be no need for the user to concern about the generated sequential code, as it is directly compilable and should not be modified. The parallel code, instead, will always have to be modified. For the understanding of the principles of operation of the TOP^2 generated code, a short description of the structure of the generated sequential and parallel code is given in the following.

8.1 Code Structure

The main tasks of the TOP^2 code generator are, first, to cut off the parallelizable program segment from the sequential program, and, second, to provide for the execution of the parallel program on the parallel machine. This is achieved by transmitting all input data from the sequential machine to the parallel machine and by returning the result data back after execution.

Assuming a principle structure of the original sequential program like the following

```
program example
   ...
  call sub(args1)
   ...
  call sub(args2)
```

```
   ...
end

subroutine sub(args)
  declarations
  executable code
end
```

the modified sequential program would basically look as follows:

```
program example
   ...
  call sub(args1)
   ...
  call sub(args2)
   ...
end

subroutine sub_org(args)
  declarations
  executable code
end

subroutine sub(args)
  declarations
  count = count+1
  if count >= start then
    call send_input_data
    if (debug) then
      call sub_org(args)
      call receive_output_data_and_compare
    else
      call receive_output_data
    endif
  else
    call sub_org(args)
  endif
  if count = stop then
    stop
  endif
end
```

In this code, the original subroutine that is to be parallelized has been cut off by simply renaming it. This renamed sequential subroutine is called if debugging is switched on in order to allow a comparison of the original sequential results with those of the parallelized routine. In both cases, with and without debugging, a modified version of the original subroutine is called that contains all declarations of the original, but none of its executable code. Instead, code has been inserted to send the input data to, and receive the output data

from the parallel machine. Furthermore, a counter has been established that counts the number of calls to the subroutine and allows to implement the semantics of the STARTSTOP directive.

The "send_input_data" and "receive_output_data" run-time routines perform two tasks, first, the distribution and merging of the data according to the user-defined distribution strategy and, second, the conversion of the data from the internal data representation of the sequential machine to that of the parallel machine and vice versa. In order to allow the implementation of TOP^2 on a broad class of sequential and parallel machines, the XDR format (External Data Representation) has been used for data conversion which is part of the NFS software and, thus, commonly available on a wide range of machines.

The basic structure of the generated parallel code skeleton is as follows:

```
    program main
      main_declarations
      call receive_input_data
      call sub(args)
      call send_output_data
    end

    subroutine sub(args)
      sub_declarations1
C--- user modifications ---
      sub_declarations2
      executable parallel code
    end

    other_subroutines
```

The parallel main program is responsible for receiving the input data and sending the output data of the parallel subroutine. It contains declarations (main_declarations) that are similar to those of the original subroutine except for distributed arrays and for arrays that are "adjustably dimensioned" in the original code. Distributed arrays have the array dimensions adjusted according to the selected data distribution policy and the size of the target processor arrays. Adjustably dimensioned arrays are implemented through dynamic memory allocation at run-time. The parallel main program (and also, of course, the parallel subroutine) is implemented in SPMD programming style, i.e. the communication with the sequential program is a parallel operation performed on all processing nodes. For communication via network this means that there is one open bidirectional socket stream for each compute node. For communication via NFS shared files it means that one NFS file is used per node and communication direction. Thus, reading and writing input and output data can be implemented through parallel I/O on all nodes.

The parallel main program normally should not be modified by the user, as modifications to it will be lost if TOP^2 is run again, generating a new version of the parallel code. Contrarily, the parallel subroutine will always have to be modified by the user and, hence, contains provisions that assure that user-specific changes are kept across calls of TOP^2.

The parallel subroutine skeleton contains practically all declarations of the original sequential subroutine (`sub_declarations1`); only the dimensions of distributed arrays are adjusted accordingly. This makes the development of the parallel code easier for the user, as argument lists, `COMMON` blocks, and type declarations look the same as in the sequential program. These automatically generated declarations may be followed by additional user-written declarations (`sub_declarations2`) and the `executable parallel code`. Furthermore, additional subroutines required for the implementation of the parallel algorithm may be added to the parallel program skeleton (`other_subroutines`). The latter three user-provided code regions are in a way protected by TOP^2 and will be kept across multiple code generation phases.

8.2 Applying Changes to the Parallel Application

The basic strategy for developing the parallel code from the given sequential program has been outlined above. If, after some tests, changes are to be applied to the existing application (e.g. changes to the data distribution strategy or the dimensions of arrays, or the parallelization of program parts different from the previously selected) certain rules should be followed. In the following, typical such modifications and how they should be handled will be described.

Changing the Number of Compute Nodes for Execution

In TOP^2 programs, the number of compute nodes used for execution will typically influence the dimensions of distributed arrays and the size of logical processor arrays and, as a consequence of this, also the distribution pattern of the distributed arrays. It is good programming practice to make all user code dependant of one or only a few symbolic constants that define the number of compute nodes used. If the actual value of such a symbolic constant is to be changed, either of two ways may be chosen:

1. Make the changes via the TOP^2 annotator's interactive dialogs. If this alternative is chosen, it is required that the program units of the application still be stored in internal format in the "`units`" directory created by the annotator. This allows to use the function "`load unit`" to read in the program unit together with all annotations and make the desired changes. Afterwards, the code generation phase has to be activated again, and, finally, the sequential and parallel program segments have to be compiled.

2. As changing the number of compute nodes does not change input and output of the parallel program segment, it is not necessarily required to run the TOP^2 annotator to make this type of changes. Instead, the changes can also be made with a text editor in the annotated Fortran program. The TOP^2 code generator, however, has to be run again after these changes in order to assure proper code generation for data distribution and array dimensioning. Note that, if changes are made using an editor, the internal representation of the program units stored in the "`units`" directory will no longer be consistent with the latest version of the annotated source code.

Changing the Distribution Strategy

In principle, both alternatives that have been described above for the modification of the number of compute nodes also apply to changes of the distribution strategy. As, however, the latter type of modification is even more incisive than the former one, it is recommended that these changes always be applied through the TOP^2 annotator. To avoid inconsistencies in the directory "`units`", no more than one application should be manipulated at a time and a new "working directory" should be established for each new application.

Parallelizing more than one Program Kernel

For the parallelization of entire sequential applications it will in most cases be necessary to parallelize more than one program kernel. In the current version, TOP^2 doesn't actively support this functionality. It is therefore recommended to proceed in steps in order to achieve the final goal. A bottom-up strategy should be followed during this process:

1. Parallelize and debug one program kernel as outlined above.
2. Store the developed parallel code (the parallelized subroutine on top and eventually additional subroutines that are needed to implement the parallel algorithm) in a separate file.
3. Clear the working directory and start the entire process of TOP^2 supported parallelization again, selecting a new program kernel from the original sequential program. Try to find a consistent strategy with respect to data distribution of global data structures. This will make the assembly of all parallelized program segments easier and more efficient.
4. After having parallelized all major program kernels, parallelize the remainder of the code. This should, ideally, be a framework consisting of components providing input data for the parallel tasks and distributing it, some sort of control flow manager that calls the parallel program segments, and eventually some postprocessing components for output.
5. Put all separately parallelized program components together to form a native SPMD parallel program. If possible, avoid the need for additional run-time support like dynamic redistribution of data. The now constructed parallel program should be completely independent of TOP^2 (perhaps with the only exception of the TOP^2 address intrinsic functions).

9 Compilation and Execution of the Distributed Application

After having made all modifications to the parallel code, the sequential and the parallel source code will have to be compiled and then executed.

9.1 Compilation

The compilation of the sequential code has to be done on the Sun workstation, the parallel code can be compiled either on the Sun, using Paragon cross-compilation, or directly on the Paragon.

For ease of use, two simple procedures are provided that allow compilation with default compiler options and assure that all required run-time libraries are properly accessed:

`top2.makes` [*name*]

`top2.makep` [*name*]

`top2.makes` compiles the TOP^2 generated sequential code named *name*`.seq.f` and must be called on the Sun. The optional parameter *name* needs not be specified if only one application is stored in the working directory (which should be the normal case). Accordingly, `top2.makep` compiles the parallel code. If it is called on Sun, cross-compilation will be used, if called on Paragon, native compilation will be used.

If the predefined compiler options used in the above procedures are insufficient, it is recommended to copy the file `makefile_sp` from `$TOP2HOME/lib` into the working directory and make the desired changes in the copy. This file is the makefile internally called by `top2.makes` and `top2.makep`, but it may as well be called directly from the command line:

`make -f makefile_sp` *target* `PRG=`*name*

name is the name prefix of the sequential or parallel code (without the trailing `.seq.f` or `.par.f`), *target* may be `seq` or `par` to compile the sequential or the parallel code, or `clean` to remove all TOP^2 generated temporary files.

9.2 Execution

After compilation, the sequential and the parallel program segments may be executed. If nothing else is specified, the names of the executables will be *name*`seq` and *name*`par`. If the selected communication method is `NFS`, both executables must be started from the NFS mounted working directory. For the sequential program normal UNIX commands may be used on the Sun workstation. On Paragon, the command

`pexec` *name*`par -sz` *size*

should be used, where *size* must correspond to the number of processing nodes specified in the distribution declarations. For either of the implemented communication methods (`NETWORK` or `NFS`) it is unimportant, which of the two programs is started first. If the selected communication method is `NETWORK`, however, the parallel program should be started no later than 90 seconds after the sequential program; otherwise the communication will be timed out.

9.3 Communication via Network

If communication via network has been switched on, either interactively or by means of the appropriate option of the code generator, all data exchange between the sequential and the parallel program is handled via UNIX socket communication. Data conversion is realized through imbedded XDR routines. Normally, the user of TOP^2 will not have to care about any technical detail of the network communication, provided that the sequential and the parallel machine are both connected to the network and the correct network name of the parallel system is specified in the resource file of the code generator (see section 4.2).

Technically, the parallel program is implemented as a communication server, the sequential program as a communication client. If the server is not yet running when the client is started, the client will wait for 90 seconds for the server to reply to the request. After this delay, the communication will be timed out. If, on the other hand, the server is started first, it will wait until the client connects. In this case, no timeout is provided. For the first handshaking between the server and the client, a fixed communication port is used that may be shared by multiple applications. This port is only used to transmit the identifiers of dynamically allocated ports that actually carry the communication for each application.

In order to make sure that a client connects to the proper server, the communication is protected by a password. The password is automatically created during code generation and is checked up when the communication is established. In rare cases, password mismatches might occur, if more than one TOP^2 application is started right at the same time. In this case, the application will stop with an error message and should be started once again. If password mismatches occur frequently, it may be useful to select a new port for the handshaking. This may be done by specifying the resource `CMD_PORT` in the resource file of the code generator as described in section 4.2.

It should be noted that, due to the password mechanism, it is not allowed to combine sequential and parallel code generated in distinct runs of the code generator.

On Sun workstations, the default limit for open file descriptors is typically 64. This will also limit the number of open socket streams to 64 or less. If more than 64 nodes are to be used in the parallel program, it will be necessary to increase the descriptor limit accordingly. The `cshell` on Sun workstations provides the command

`limit descriptors` *nnn*

that sets the current limit to *nnn* open descriptors.

9.4 Communication via NFS Files

In the current version of TOP^2 the communication between Sun and Paragon may also be realized through NFS shared files. To allow for parallel I/O on Paragon, one file is used per compute node for each communication direction. If not specified otherwise, the input data for the parallel program segment is sent in files named `IN.`*nnn*, the output is returned in files named `OUT.`*nnn*, where *nnn* is a three-digit suffix defining the number

of the compute node the data is related to. For synchronization between the sequential and the parallel program segments, two files `lock0x0.file` and `lock0x1.file` are used. Normally, these files will be invisible to the user, if, however, the distributed application is killed during execution, it may be necessary to explicitly delete them.

9.5 Rerunning the Parallel Program with Existing Data

In certain situations, it may be useful to rerun the parallel program on Paragon without also rerunning the sequential code and, thus, reuse the already existing input data of a previous run. This mode of operation is possible if communication is via NFS files, if the sequential and the parallel program segments have already been run successfully, and if the input data produced by the sequential program still exist in the NFS files.

To start the parallel program in this mode, it is necessary to manually set up the synchronization files with the commands

```
rm lock0x0.file
touch lock0x1.file
```

and then start the parallel program as described above.

This will initialize the synchronization files for one NFS read/write cycle, i.e. the parallel subroutine will be called exactly once, read the input data, produce new output data, and then stop. Note, that in this mode of operation no debugging of the output data will take place even if debugging is switched on, as the parallel program runs unattended by the sequential program.

9.6 Debugging

If debugging is switched on, the control and data flow during execution of the distributed application are slightly different from the normal case (Figure 5). With debugging switched on, not only the parallelized program kernel is executed, but also the original sequential kernel. After execution, the results of both components (all variables that are declared to be output) are compared. Integer, Logical, and Character variables are compared for identity, Real, Double Precision, and Complex variables are compared for equality within a certain allowed relative absolute error. Approximately two deviating digits are allowed due to round-off errors.

If the results of the sequential and the parallel program kernel do not match, the TOP^2 run-time system will display messages on the Sun display giving the name of the involved variable (for arrays also the global array index), the node number, and the computed result of the sequential and the parallel program kernel.

If non-distributed arrays or variables are declared to be output, TOP^2 checks if all nodes redundantly return the same result. If not, a warning message is displayed. In either case,

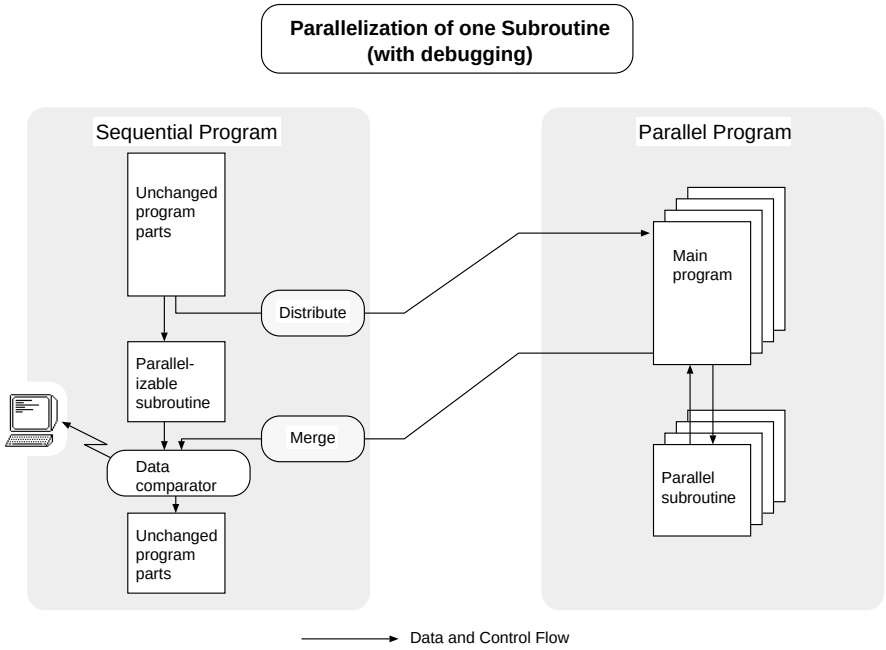

Figure 5: Control Flow and Data Flow with Debugging

only the result of node zero is copied back to the sequential program.

10 Example

10.1 Matrix Multiplication

In the following, all major steps required to parallelize a given sequential program for Paragon using TOP^2 as a parallelization aid shall be demonstrated in an example. The example program implements a matrix multiplication and contains the following components:

`mxm.f`: Source of the sequential program.

`mxm2.p`: Parallel implementation of the matrix multiplication kernel (subroutine `MATMUL`).

`mxm2e.p`: Parallel matrix multiplication kernel with imbedded error for demonstration of debugging.

The sources of the example program components are stored in `$TOP2HOME/demo`.

Step 1: Annotation and Code Generation

The sequential program contains the subroutines `INIT` for the initialization of matrices `A`, `B`, and `C`, `MATMUL` for the matrix multiplication, and `CHKSUM` to compute a checksum of result

matrix A. The parallel code for subroutine MATMUL implemented in file mxm2.p is based on a distribution of all three basic arrays A, B, and C in blocks of multiple columns and requires that the corresponding array dimensions be divisible by the number of processing nodes.

In the following, a distribution onto 8 nodes is used. Calling the TOP^2 interactive annotator and executing the interactive steps as described in section 2 should result in the following directives inserted into the sequential program (file mxm.top2.f in the working directory):

```
CKFA$ DEBUG

CKFA$ PROCESSORS P(IP)

CKFA$ IN B,C,M,N,L
CKFA$ INOUT A
CKFA$ DISTRIBUTE A(*,BLOCK) ONTO P
CKFA$ DISTRIBUTE B(*,BLOCK) ONTO P
CKFA$ DISTRIBUTE C(*,BLOCK) ONTO P

      INTEGER IP
      PARAMETER(IP=8)
```

In the above, the IN and INOUT directives reflect the results of the interprocedural data flow analysis. B and C are input arrays used as operands for the matrix multiply, the scalar variables M, N, and L are used for the adjustable dimension of all arrays and also as loop boundaries and are therefore input to the parallel program kernel. Array A is input and output, as the initialization values are carried from subroutine INIT into subroutine MATMUL and the results are transferred back to subroutine CHKSUM. A parallel implementation of subroutine MATMUL, however, including the initialization of matrix A in the parallel code, would allow to change the INOUT directive into an OUT directive.

According to the given parallel algorithm, data distribution is done in the second dimension using BLOCK distribution strategy for all three arrays. The blocks of data are distributed onto the one-dimensional logical processor array P of size IP. A PARAMETER statement has been inserted by the TOP^2 annotator to assign a specific value to the symbolic constant IP. Note that distribute directives without the ONTO clause would lead to essentially the same data distribution onto the one-dimensional default processor array. The size of this array would, however, only implicitely be given through the respective TOP^2 default, as no other processor array definitions are present in this code.

During interaction with the TOP^2 annotator, debugging has been switched on, which results in the corresponding DEBUG directive.

The resulting generated sequential and parallel code are stored in files mxm.seq.f and mxm.par.f (see functions "NFS Transform" and "MP Transform" in section 3).

Step 2: Implementation of the Parallel Algorithm

The generated sequential code for our example is complete in that it may be compiled without further changes. The generated parallel code skeleton comprises a main program and the declarations of the parallel subroutine `MATMUL`. The main program contains calls to the cross domain message passing routines receiving the input data and returning the output data of subroutine `MATMUL`; it should not be changed. The parallel subroutine `MATMUL` is ready to take up the parallel code. Any changes made to the subroutine should be inserted between the comment line

```
C --- MATMUL: --- Make all your code changes below this line ---
```

and the `END` statement of the subroutine. In our example, it is sufficient to include file `mxm2.p` here.

Additional subroutines required for the implementation of the parallel algorithm could be added after the comment line

```
C --- Add all new subroutines below this line ---
```

According to the data distribution policy specified for this example, each node program of the parallel SPMD code receives one block of columns of matrices `A`, `B`, and `C`. Thus, the distribution of these arrays determines the loop bounds of the loops in the parallel code.

In the example code, the TOP^2 intrinsic function `TOP2_GET_SHAPE` is used to compute the array bounds of arrays `A` and `B`. In cases where explicit computations are to be made depending on the number of processors used, the TOP^2 intrinsic function `TOP2_GET_PSHAPE` has been used.

Step 3: Compilation and Execution

Compilation of the sequential and the parallel program is straightforward now and can be done by using the procedures

```
top2.makes
top2.makep
```

that may both be called on the Sun, if cross-compilation is to be used (otherwise `top2.makep` must be called on Paragon).

After this, executables named `mxmseq` and `mxmpar` will have been created in the working directory (compare section 2). On the Sun, the sequential program may be executed by the command

```
mxmseq
```

the parallel program may be executed on Paragon using the procedure

```
pexec mxmpar -sz 8
```

Note that it is not important which of the two programs is started first. If communication is via NFS files, however, both must be started from the NFS mounted working directory.

The resulting output on Paragon should look similar to

```
 Starting matmul ...
 dclock:   0.7763022799990722
FORTRAN STOP
```

on Sun the output will be

```
Calling matmul ...
Starting matmul ...
Result =   -10035482.5739520
```

Note that, due to debugging being switched on, the matrix multiplication kernel will redundantly be executed on the Sun workstation also.

Step 4: Debugging

To test debugging, the parallel code included in file `mxm.par.f` may now be deleted again, and the file `mxm2e.p` be included instead. The parallel code has to be compiled again and executed together with the sequential code.

The output obtained on the Sun workstation should now look as follows:

```
Calling matmul ...
Starting matmul ...
DEBUG: wrong results in variable A of subroutine MATMUL on node   5:
  Element(  2  94) Seq:   -97.714313400000 Par:    8.5000000000000
Result =   -10035376.3596386
```

The index (2,94) of the erroneous array element is global and translates to local index (2,4) on physical node 5 according to section 6. The correct result for this array element is given by the term "`Seq: -97.714313400000`", the result received from the parallel program is instead "`Par: 8.5000000000000`".

10.2 Jakobi Solver for Poisson Equation

The directory `$TOP2HOME/demo` contains a second, more comprehensive example for the solution of Poisson equations with Jakobi relaxations. The example consists of four files named

```
jakobi.f
jakobi.top2.f
jakobi.par.f
jakobi.README.
```

The file `jakobi.README` contains all information required to run the example with TOP^2.

11 Limitations

Currently, some limitations are to be considered when using TOP^2. The most important ones are listed below:

1. TOP^2 accepts ANSI Fortran 77 programs that may contain language extensions like
   - long names
   - underscore characters in names
   - `DO / ENDDO`
   - lower case source
   - *2, *4, *8 etc. notation for type lengths
   - `IMPLICIT NONE`
   - `NAMELIST`
   - `TASKCOMMON`

   However,
   - array syntax
   - pointers
   - 128 bit double precision data type

   are currently not supported.
2. TOP^2 requires that the entire input program be contained in one file and all includes and preprocessor macros be resolved.
3. Normally, TOP^2 generates all declarations required for the parallelized program segment. However, currently DATA statements generated in the parallel code segment for distributed arrays do not reflect the chosen data distribution scheme. Thus they have to be adjusted manually.
4. Currently, TOP^2 does not support adjustable dimension CHARACTER arrays as well as assumed size arrays in the parallel code segment, as the proper dimensions cannot be determined at run-time.

12 TOP^2 Installation on Sun and Paragon

The current implementation of TOP^2 requires several components - executable modules and procedures, run-time libraries, resource files, documentation, and examples for demonstration purposes. The diagram in Fig. 6 shows the placement of all TOP^2 files on Sun and Paragon, and also those files the user may have to deal with in the working directory.

13 Specific Considerations for the T3D Version in Eagan

Due to the specific situation, some details of the installation and usage of TOP^2 on the Cray T3D system at CRI in Eagan differ from the description given in this manual. In the following, items specific to the T3D version are described.

13.1 Installation

TOP^2 is installed on T3D system `typhoon` in directory

```
/cray/uss/u6/n5006/top2
```

All files are world accessible. To use the TOP^2 tools and scripts the PATH environment variable should include the directory

```
/cray/uss/u6/n5006/top2/bin
```

and the environment variable `TOP2HOME` should be set as follows:

```
TOP2HOME = /cray/uss/u6/n5006/top2
```

In case the directory `top2` is to be copied to another place, the path and the environment variable have to be adjusted accordingly.

13.2 Resources

The TOP^2 code generator `top2pp` requires a resource file that specifies default values for some code generation options. The default resource file is

```
$TOP2HOME/resources/top2pprc
```

To make a copy of this file to another place, the instructions in section 4.2 should be followed.

The TOP^2 Annotator requires an X-based resource file. To assure that the resources are properly activated, they should be explicitly loaded into the X server's data base:

```
xrdb $TOP2HOME/resources/Top2
```

13.3 Files

As described previously, it is best to use an NFS mounted file system for the Fortran sources that are to be used with TOP^2. This file system should be accessible by the workstation where the TOP^2 tools run and also by the T3D host and the T3D itself.

Normally, TOP^2 generates all files in this file system, so they can directly be compiled with the scripts `top2.makes` and `top2.makep`.

As, however, the T3D loader currently doesn't load binaries from NFS files, the compiled binaries are currently copied to `/ptmp/$LOGNAME`. Hence, the execution of the sequential and the parallel code should be started from this directory. Furthermore, if communication is not via UNIX sockets but via NFS files, these files will as well be put into `/ptmp/$LOGNAME`, if the binaries were started there. Changes to these file naming conventions can be made in the scripts `top2.makes` and `top2.makep`.

Figure 6: TOP^2 Files